<u>Computer Vision</u>

CS – GY – 6643

Project Report - 2

- **Correlation/ Convolution**
- **Cross – Correlation and Template Matching**
- **Creative Part**

Rhuthvik Dendukuri – rd3377

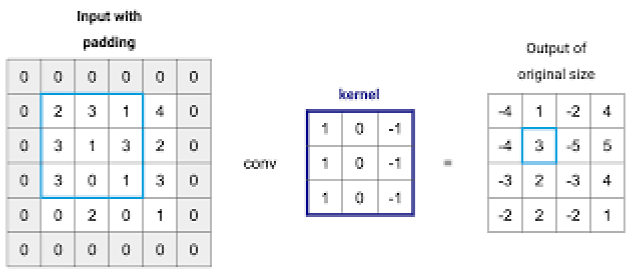
3/17/2024

Table of Contents:

1. Correlation/ Convolution:

1.1Convolution and its working:

Convolution is a fundamental operation in image processing and computer vision. It involves applying a mathematical operation between two functions, typically an input image and a filter or kernel. The convolution operation produces a new image, known as the output or feature map, by sliding the filter over the input image and computing the weighted sum of the overlapped pixel values.



- **Kernel or Filter**: A small matrix of numbers defining the operation to be applied to the input image.
- **Sliding Window Operation**: The kernel is moved over the input image, and at each position, the corresponding pixels are multiplied and summed to produce a value for the output image.
- **Element-wise Multiplication and Summation**: The core of the convolution operation involves element-wise multiplication and summation between the kernel and the input image region.
- **Boundary Handling**: Special handling is required for convolution at image boundaries to maintain accuracy, often achieved through padding techniques like zero-padding or symmetric padding.
- **Output Image Size**: The size of the output image depends on the size of the input image, kernel size, and padding. Typically, the output image is smaller than the input due to the convolution operation.
- **Applications**: Convolution finds applications in various image processing tasks such as blurring, edge detection, noise reduction, and feature extraction, utilizing different types of filters like Gaussian, Sobel, and Laplacian filters.

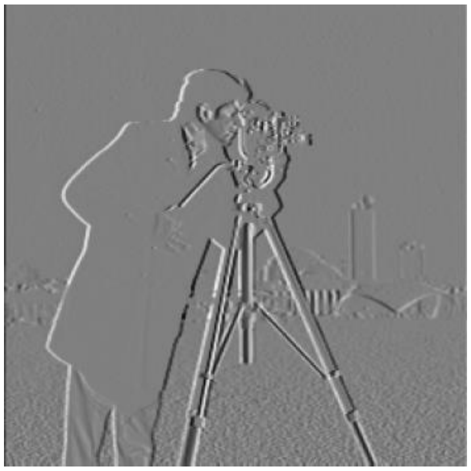
1.2 Edge Detection Filtering

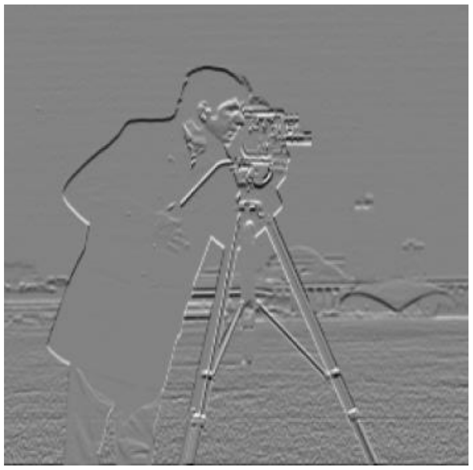
Cameraman.png

zebra.png

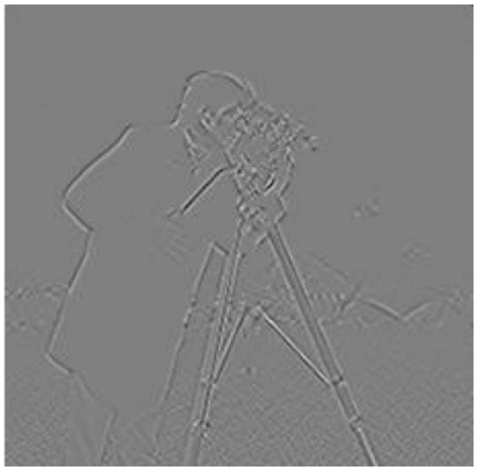
Convolution:

Without the usage of pre-built functions, a unique 2D convolution module has been constructed. This module receives two 2D arrays as input—a filter, f, and an image, I—and outputs their convolution, f∗I. Based on the size of the filter f, zero-padding has been used to alleviate boundary issues within image I. Additionally, the module supports masks that serve as image templates for later jobs as well as 1D line filters, which are used for separable filtering.


Horizontal Edges with 1D filter


Vertical Edges with 1D filter

With a horizontal filter of $[-1\ 0\ 1]$, we convolve the given "**cameraman"** image to find the horizontal edges of the image. Assume the image's named as "**cameraman_hor_edge**". Similarly, with a vertical filter of $\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$, convolving with the image "**cameraman"**, we get the vertical edges. Convolving the "**cameraman_hor_edge**" with the above vertical filter, the result image will be showing all the possible image edges.



Edge Detection using 1D filter on Cameraman image

Performing the same operations on a "**zebra**" image.

Horizontal Edges with 1D filter

Vertical Edges with 1D filter

Edge Detection using 1D filter on Zebra Image

1.2.1 Version via First Derivatives:

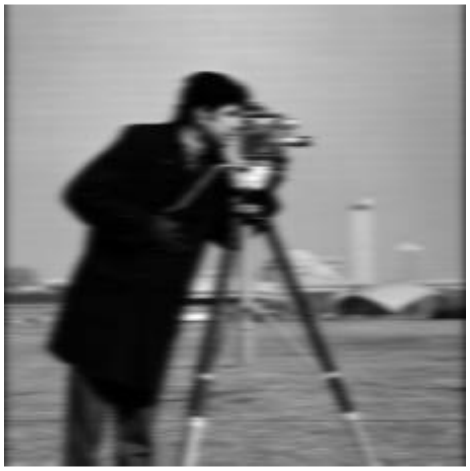
For Edge detection using first derivatives, we follow the below procedure:

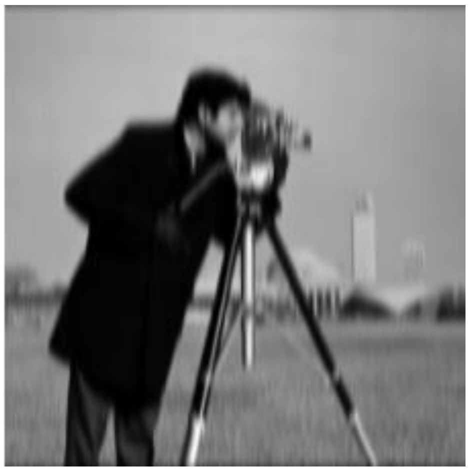
Considering Gx and Gy as the Gaussian 1D filters for horizontal and vertical edge detections respectively.

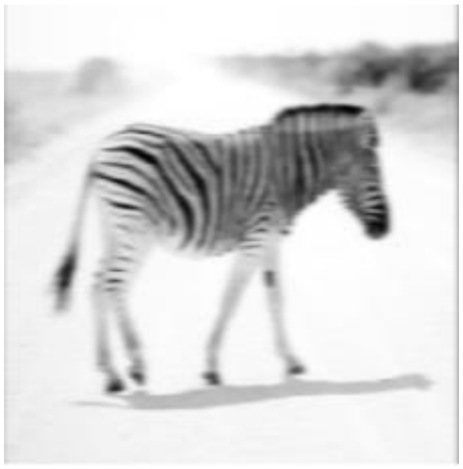
If Gx and an image are convolved, we get an image that is blurred horizontally. In the same way, if Gy and an image is convolved, we get an image that is blurred vertically. These can be considered as 1D filtered images.

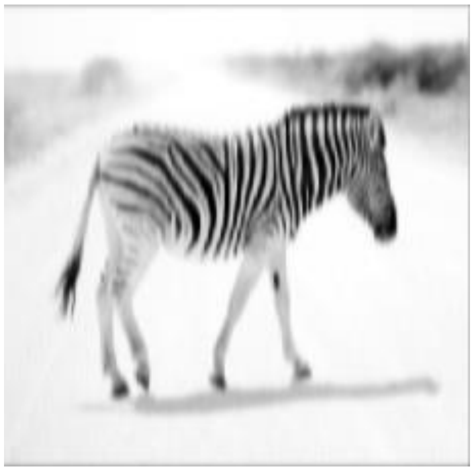
$GaussImg(x,y)_x = [Gx] * Image(x,y)$ would give a horizontal blur of the image.

$GaussImg(x,y)_y = [Gy] * Image(x,y)$ would give a vertical blur of the image.


Horizontal Gaussian Blur on Cameraman image


Vertical Gaussian Blur on Cameraman Image


Horizontal Gaussian Blur on Zebra Image


Vertical Gaussian Blur on Zebra Image

For 2D filtered images, we need to combine convolution of Gx and Gy filters on the image.

This is given by,

$$GaussImg(x, y) = [Gx] * ([Gy] * Image(x, y)]$$

The images would look blurred on the whole.

Gaussian Blurred Image after 2D filtering on Cameraman image

Gaussian Blurred Image after 2D filtering on Zebra image

After the filtering is done, we compute the image gradient and gradient magnitude of the filtered image by the following:

$$xderiv = [-1\ 0\ 1] * \ GaussImg(x, y)$$

$$yderiv = [-1\ 0\ 1]^T * GaussImg(x, y)$$

$$GradMag = sqrt(xderiv^2 + yderiv^2)$$

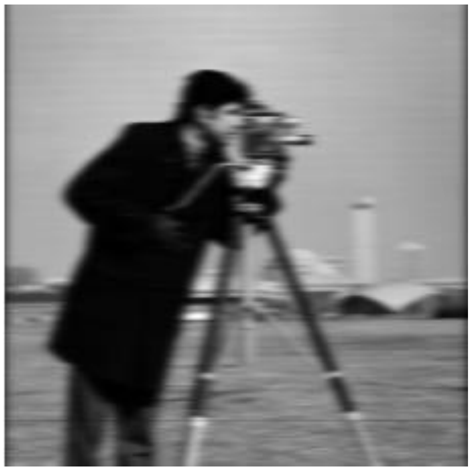
This gradient magnitude image is thresholded as per demand to get the strongest edges of the image.

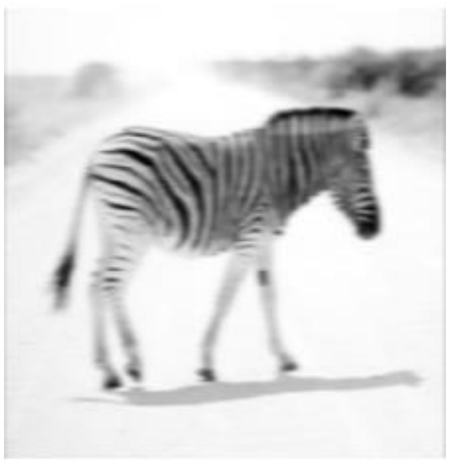
1.2.1.1 Gaussian Smoothing:

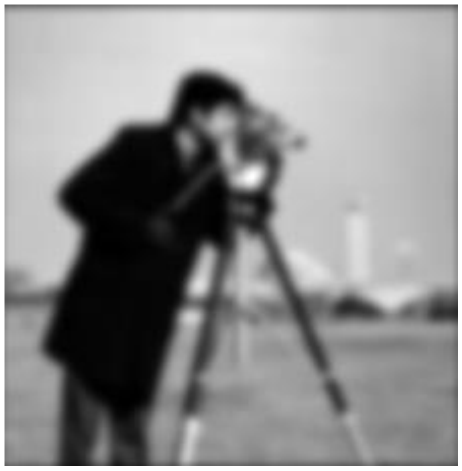
Gaussian smoothing is first applied using 1D separable Gaussian filters. By representing the 2D Gaussian filter as the sum of two 1D Gaussian filters, its separability is shown.

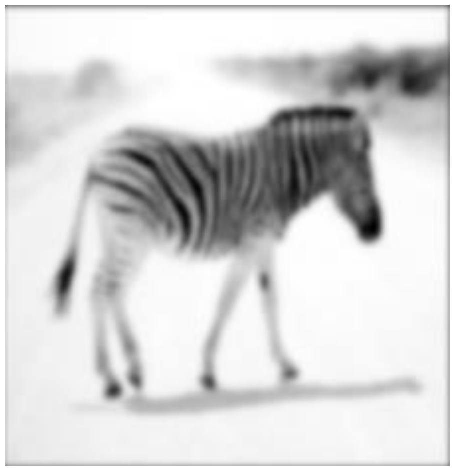
Two parameters are needed to construct the Gaussian kernel programmatically: the kernel size (6*sigma+1) and the sigma value (3.0). After the kernel building, the resulting kernel is used to apply Gaussian blurring to the image. The picture is convolved along one dimension at first, and subsequently along the other, in order to apply 1D Gaussian filters. The blurring effect that results is comparable to using a 2D filter.

Based on below Gaussian image results, it is evident that some degree of blurring has occurred, a result influenced by the kernel size and sigma values provided as the Gaussian parameters.


1D Gaussian Filtering

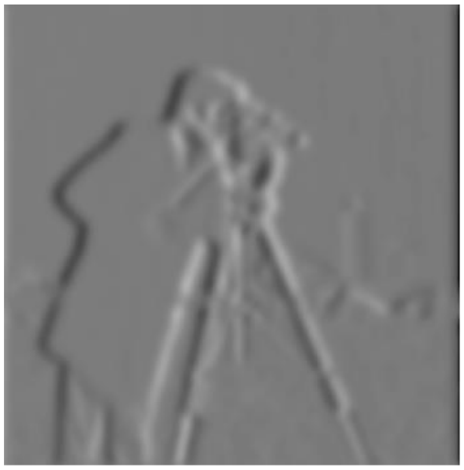

1D Gaussian Filtering

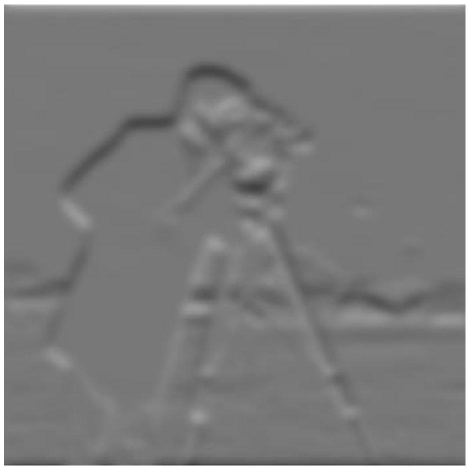

2D Gaussian Filtering

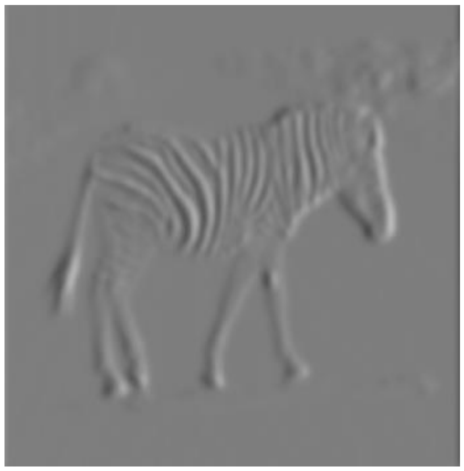

2D Gaussian Filtering

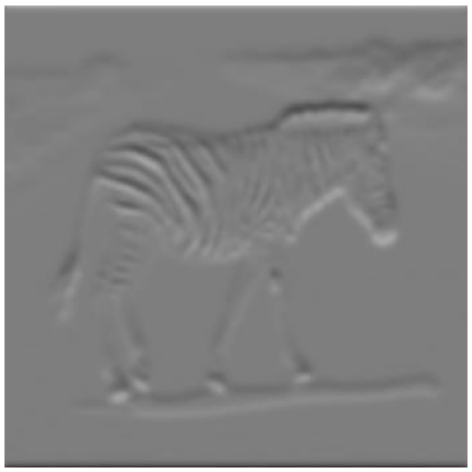
1D X and Y Derivatives:

The next step involves extracting the x and y derivative pictures from the Gaussian-blurred images by applying 1D derivative filters to them. The horizontal direction of these derivative filters is set to [-1,0,1], and the vertical direction is set to its transpose. The derivative filters are applied to the Gaussian-blurred pictures after the convolution step described in the preceding Edge detection approach.


Horizontal Edges of Gaussian Blurred Cameraman Image


Vertical Edges of Gaussian Blurred Cameraman Image

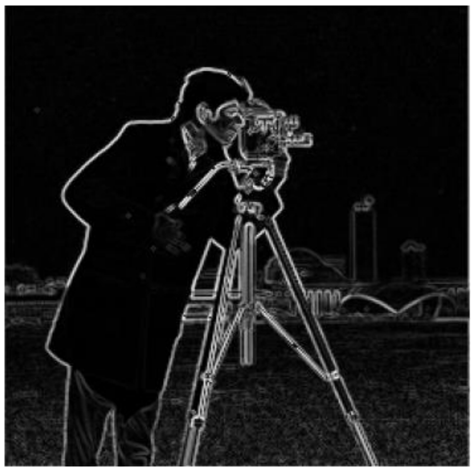

Horizontal Edges of Gaussian Blurred Cameraman Image

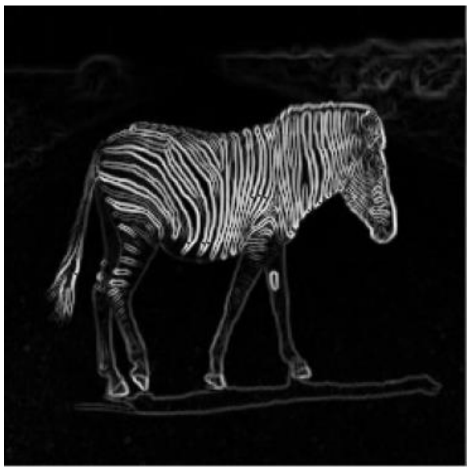

Vertical Edges of Gaussian Blurred Zebra Image

Based on above image results, it is evident that the x-derivative filter accentuates image edges in the vertical orientation, while the y-derivative filter emphasizes image edges in the horizontal orientation.

Gradient Magnitude:

The next step in the edge detection process is to create the gradient magnitude image using the x and y derivative images that were previously shown. This is done by using the formula that was described in the Edge detection approach, $\sqrt{xderiv^2 + yderiv^2}$.

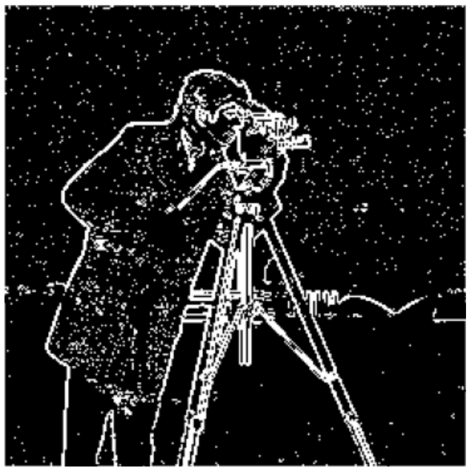

Gradient Magnitude Image of Cameraman Image

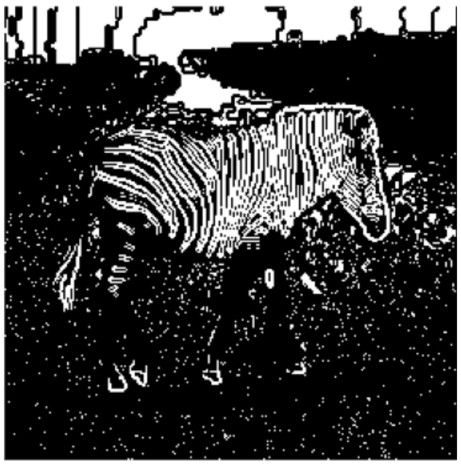

Gradient Magnitude Image of Zebra Image

Above outcomes are obtained after applying the gradient magnitude function to the derivative images.

Threshold:

To highlight edges, the gradient magnitude images are thresholded as the last step in the edge identification process. Otsu Thresholding has been used to find the ideal threshold. Otsu determines the ideal threshold setting for the supplied gradient image automatically.


Otsu Thresholded Gaussian Cameraman Image


Otsu Thresholded Gaussian Zebra Image

Presented above are the final outcomes of the edge detection algorithm post-thresholding.

1.2.2 Edge Filtering via Zero-Crossings of Second Derivatives:

For Edge detection using zero-crossings of second derivative we follow the below procedure:

- Initially, take the Gaussian-smoothed image results obtained earlier, where noise has been reduced through the application of separable 1D Gaussians:

$$GaussImg(x,y) = [G1x] * ([G1y] * Image(x,y))$$

- Then, apply a 3x3 Laplacian filter to mimic the 2nd derivative in 2D: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

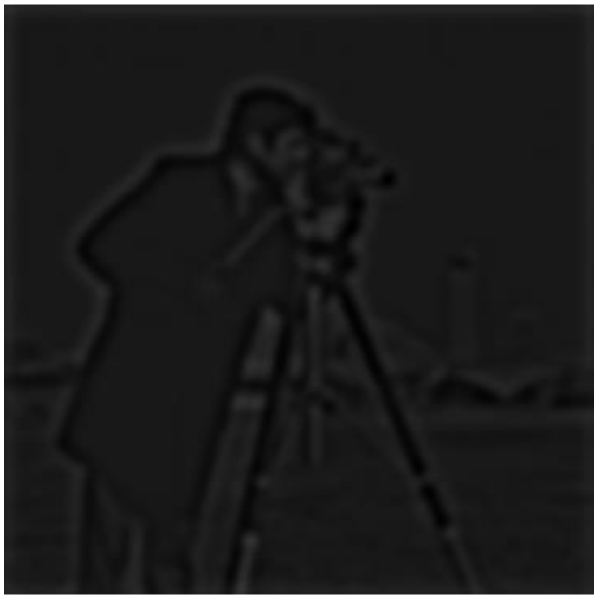
- Finally, apply a 3x3 procedure for detection of zero-crossings by adhering to the principle of "detecting pixels with positive values that possess at least one negative neighbor".

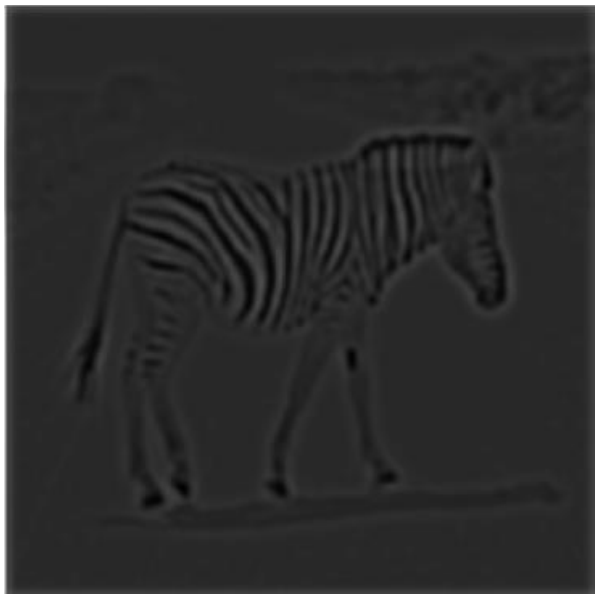
Gaussian Smoothing:

From the above computed Gaussian smoothed images, we perform convolution with a Laplacian filter.

3x3 Laplacian Filtering:

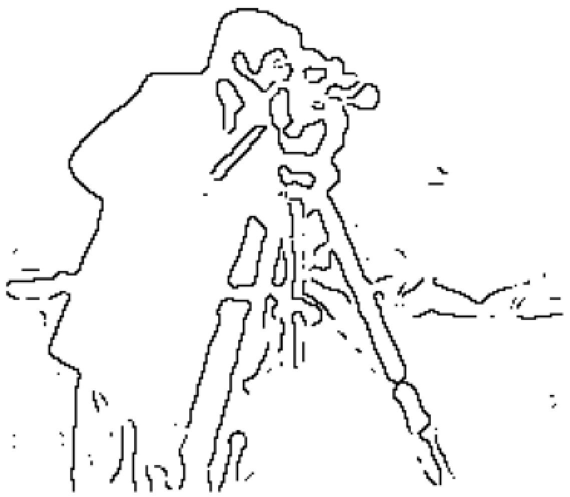
The subsequent phase entails the utilization of 3x3 Laplacian filters on the Gaussian-blurred images. This filter is predefined as per assignment requirements, set to: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ in 2D.

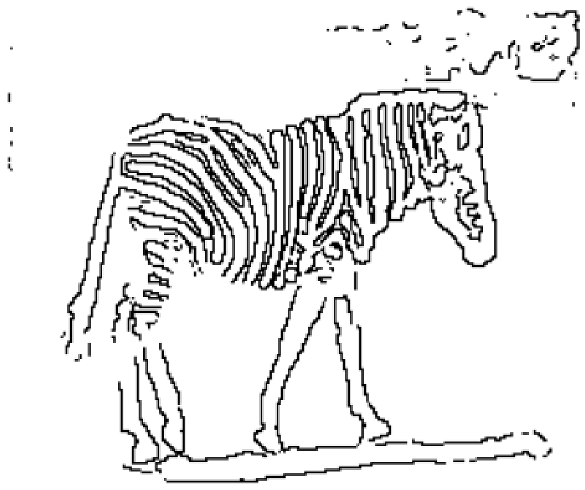

Laplacian of Gaussian of Cameraman image


Laplacian of Gaussian of Zebra image

Detection of Zero-Crossings:

We scan every pixel in the Laplacian-filtered, Gaussian-smoothed image that has a positive value. Examine all eight of the nearby pixels for each of these types of pixels to see if any of them have negative values. If at least one of the adjacent pixels in the output image is negative, then that pixel is considered an edge. To capture only the strongest of the edges, another minor threshold is used.


Zero Crossings of the Cameraman image


Zero Crossings of the Zebra image

The resulting images captures all edges, including those within noisy regions.

2. Cross-correlation and Template Matching

2.1 Correlation and its working:

Image correlation is a method used in computer vision and image processing to calculate how similar two images are, or how distinct areas within the same image are from one another. It entails calculating a numerical measure of similarity by comparing the intensity values or patterns in corresponding pixels or regions of the images. Image correlation's fundamental idea is to slide one image—referred to as the template or kernel—over another—referred to as the reference or search image—while calculating a similarity measure at each location. In many different applications, including object detection, motion tracking, picture registration, and stereo vision, image correlation is frequently employed. It makes it possible to locate items or patterns in photos, follow their movements, align photos for more examination, and spot adjustments or variations.

- **Correlation Operation**: Correlation is another fundamental operation in image processing and computer vision. It involves measuring the similarity between two functions, typically an input image and a template or kernel. Unlike convolution, correlation does not involve flipping the kernel.
- **Kernel or Template**: Similar to convolution, a kernel or template is used in correlation to define the operation. However, unlike convolution, the kernel is not flipped during correlation.
- **Sliding Window Operation**: Correlation also employs a sliding window operation, where the kernel is moved over the input image. At each position, the corresponding pixels are multiplied and summed to produce a value for the output image.
- **Element-wise Multiplication and Summation**: Similar to convolution, the core of the correlation operation involves element-wise multiplication and summation between the kernel and the input image region. However, in correlation, the kernel is not flipped.
- **Boundary Handling**: As with convolution, special handling may be required for correlation at image boundaries to maintain accuracy. Various padding techniques can be employed to handle edge cases.
- **Output Image Size**: Like convolution, the size of the output image in correlation depends on the size of the input image and the kernel. Typically, the output image has dimensions similar to the input image.
- **Applications**: Correlation is used in various image processing tasks such as template matching, pattern recognition, and object detection. It is particularly useful in locating instances of a template image within a larger image or in detecting specific patterns or features.

2.2 Correlation and Convolution:

In the case that the kernel is symmetric about the axes, the two operations—correlation and convolution—are identical. Cross-correlation is a commonly used statistical tool for determining the degree of similarity between two signals. On the other hand, convolution functions as a filtering process that is applied to a signal, changing its properties according to predetermined parameters or kernels. Despite having different purposes, these two processes are essential procedures that are widely used in many different areas of image processing and related applications.

Both 1-D and 2-D masks are possible Mathematically, the correlation operation can be represented as:

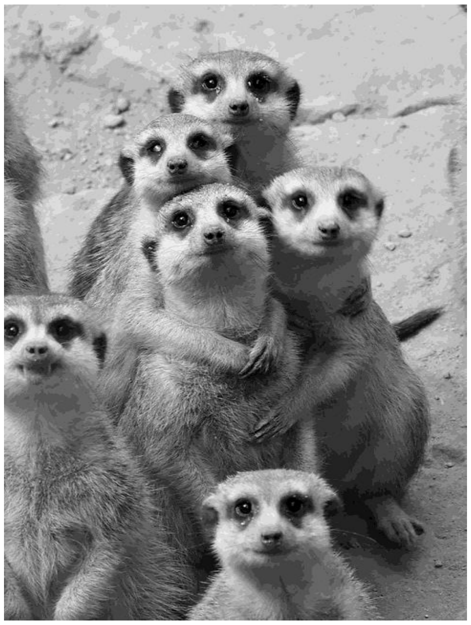
1D $$g(i) = \sum_{k} f(i+k)h(k)$$

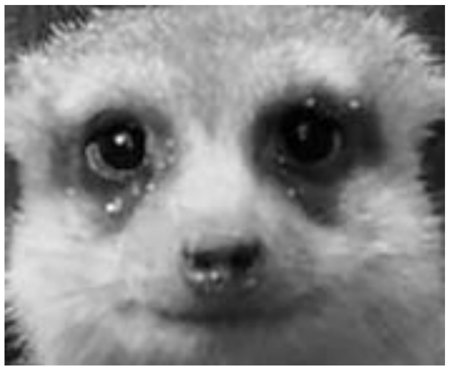
2D $$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

$$\boxed{g = f \otimes h}$$

$\otimes$: correlation operator

2.3 Applying Template Matching on 'animal-family-25.jpg'


Animal-family-25.jpg


Animal-family-template-25.jpg

Template Matching:

In image processing and computer vision, template matching with correlation is a technique used to identify particular items or patterns within a broader image. First, a template image that depicts the desired pattern is chosen. This template is slid across various portions of the bigger image, and a similarity measure is computed, usually using cross-correlation or normalized cross-correlation. This metric expresses how closely each overlapping area of the image matches the template. Peaks in the matching score serve as potential matches by showing where the template most closely resembles a certain area of the image. The system locates occurrences of the template inside the enlarged image by identifying these peaks, making it possible to identify specific items or patterns.

The steps listed below are what we do for Template Matching:
• Setting up the template image by calculating its mean, maximum, and minimum values. Next. To ensure that

the final template has a mean value of 0.0, normalize the template image by deducting the mean value from each pixel value.

• Using the zero-mean template picture as a mask, perform cross-correlation between the original and template images.

• When some areas of the image closely match the template, this procedure produces peak values, or maxima. The modified correlation image is thresholded at 60% of 255 (150), producing peaks for every match in the set. In order to make visual inspection and verification easier, peaks are superimposed onto the original image.

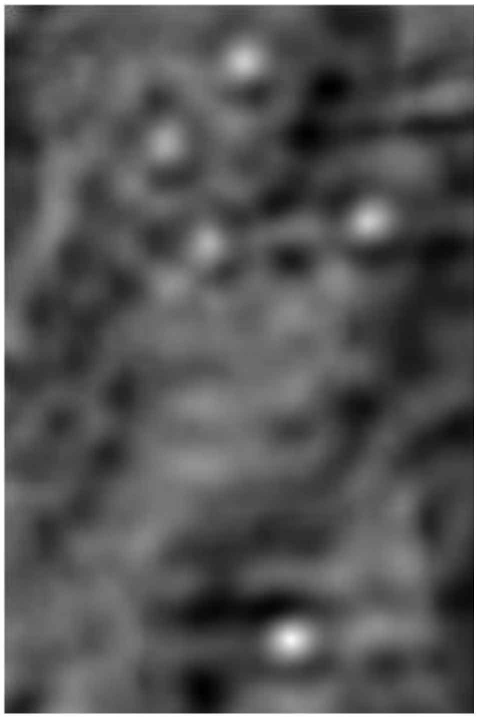
Zero-Mean-Template:

To obtain the zero-mean template, the template must first be normalized. We calculate the template image's mean, minimum, and maximum intensity values. Then, we subtract the template's mean value from each pixel in the template image to create the zero-mean template.

Animal-family-template-25.jpg

Cross – Correlation:

Cross-correlation between the original image and the zero mean template image is the next step in the process. The image is padded in accordance with the template size to guarantee compliance. (Image height - template height) divided by (Image width - template width) yields the correlation image's size. Then, cross-correlation is run between the template and any area where the image overlaps.

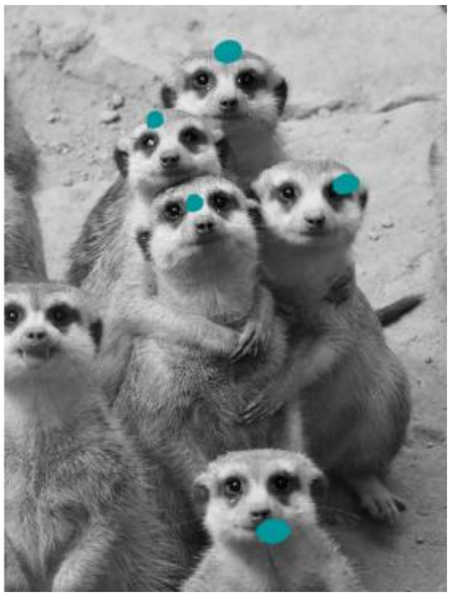


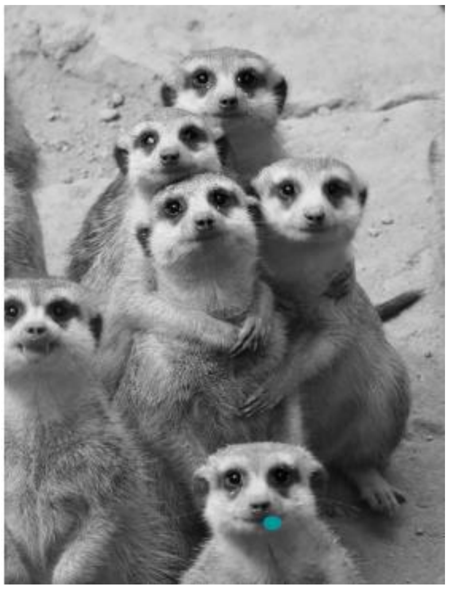
Correlated image of Animal family

Original Animal family Image

Thresholding and peaks generation:

The process ends with the application of a threshold and the identification of peaks. As the template gets closer to a specific overlapped location, the intensity value in the cross-correlation output image rises. As a result, the image's peak values indicate the best fit for the template in the provided image. These peak values are located by scanning the correlation image, which is above the threshold of 60% of 255. To aid in visual inspection and verification, the locations found after the scan are superimposed onto the original image. Several other thresholds are experimented and documented as given below.


Findings when Threshold is 60%


Findings when Threshold is 80%

Findings when Threshold is 100%

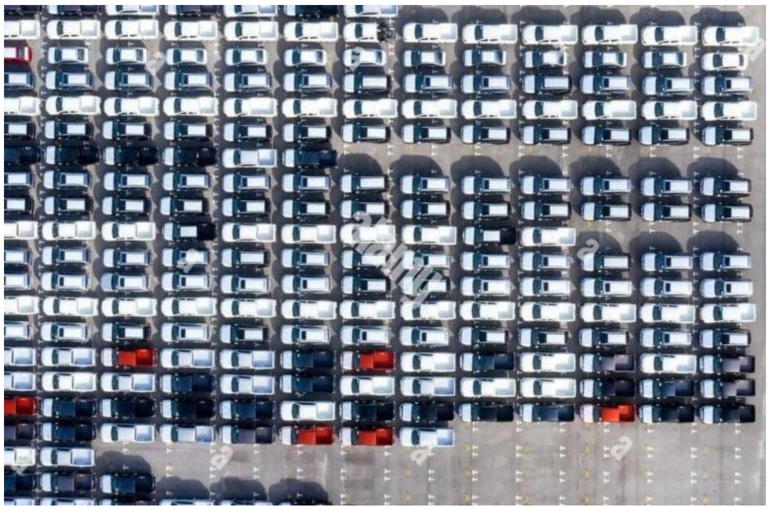
Experiment:

Given a threshold of 60%, the resulting image had most of the similar templates in the input image. As and when the threshold is increased, the strictness of the comparison is increased as well. When the threshold is 80%, the exact template is pointed out as the input template. In addition, when the threshold is at 100%, there is no match for the input template in the input image.

Threshold	**Number of Matched templates**
60%	5
80%	1
100%	0

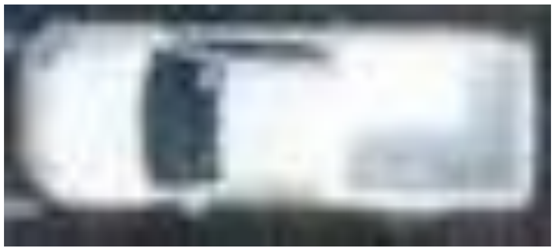
3. Creative Part:

In this case, we take a similar strategy to that used earlier, using Template Matching to ascertain how many vehicles of a particular kind are parked in the lot. To ensure that the final template has a mean value of 0.0, the automobile template image is first prepared and normalized by deducting the mean value from each pixel value. Next, using the zero-mean car template picture as a mask, cross-correlation is done between the original parking lot image and the image. When particular areas of the image are closely aligned with the template, this process produces peak values, or maxima. Peaks are found over the whole set of matches by thresholding the adjusted correlation image. The thresholded image's contours are then created around these peaks, and the number of contours represents the number of vehicles of the designated kind in the parking lot. To aid in visual inspection and verification, the peaks are also superimposed into the original image.

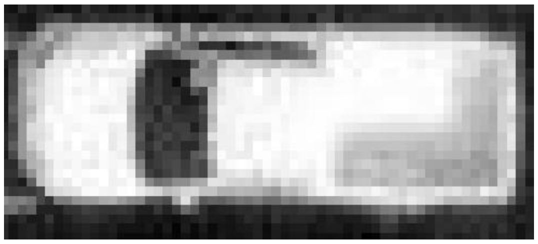


Car Parking lot

Zero - Mean - Template:



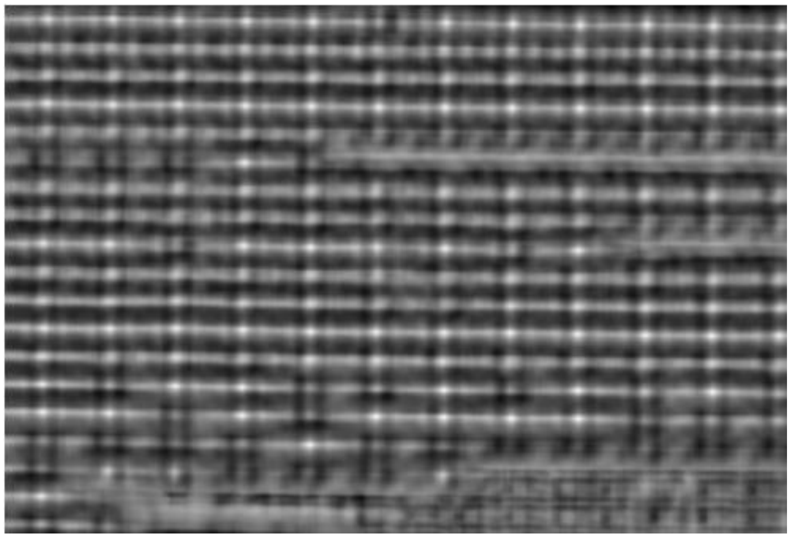
Car template



Zero Mean Car Template

Cross - Correlation:

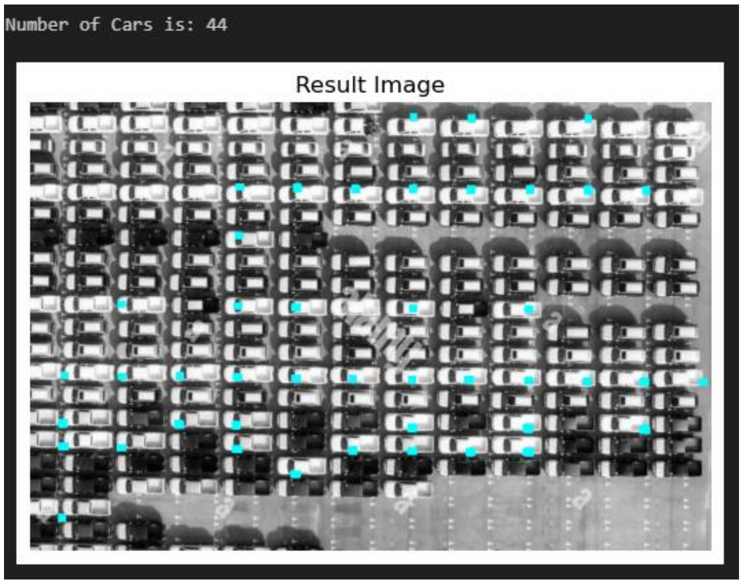
The next step involves conducting cross-correlation between the parking lot image and the zero-mean car template image. To ensure compatibility, the image is padded to match the template size. Following this, cross-correlation is executed between every overlapping region of the image and the template.



Cross - Correlation between original car parking lot image and zero mean car template image

Thresholding, peak generation and contouring:

The process ends with the application of a threshold, the formation of contours, the identification of peaks, and their counting. The intensity value in the cross-correlation result image rises as the template more closely aligns with a certain overlapping region, and as a result, the image's peak values represent the best possible matches. The correlation image is scanned in order to find these peak values that are higher than a set threshold. The generated image then has contours defined around these peaks, and the count of these contours represents the quantity of the designated type of cars in the parking lot. Moreover, to aid in visual inspection and verification, the regions that the scan identified are superimposed into the original image.



```
Number of Cars is: 44
```

The car template matched with 44 cars in the given input car parking lot image

From the result, we can see that there is a total of 44 cars that are similar to the car in the chosen template.

4. Appendix

The code was run on a Jupyter notebook file with the python kernel version being 3.11.5.

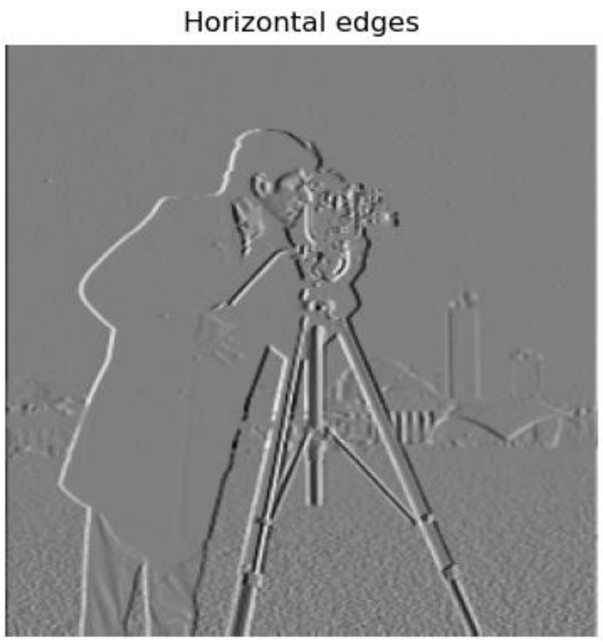
```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
import math
def read_image(path):
    original_image = Image.open(path)
    resized_image = original_image.resize((256, 256))
    im = np.array(resized_image.convert('L'))
    im_uint8 = ((im - np.min(im)) * (1/(np.max(im) - np.min(im)) *
255)).astype('uint8')
    return im_uint8
```

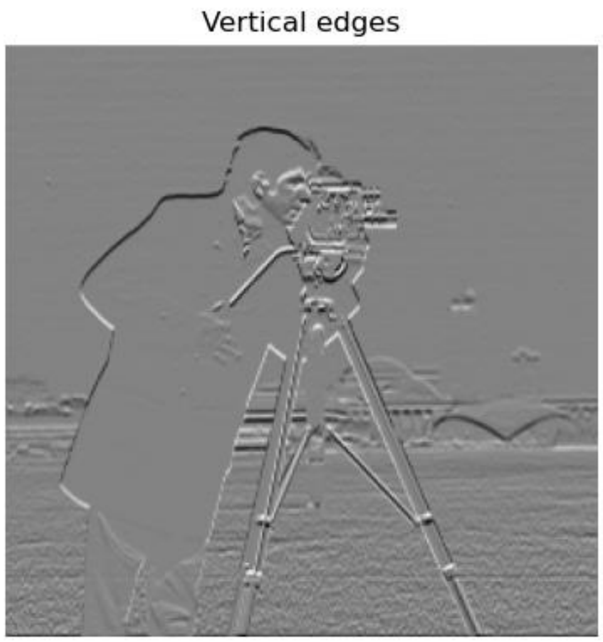
```python
def convolution(f, I):
    filter_ht, filter_wdt = f.shape
    # padding = filter_ht - 1
    image_ht, image_wdt = I.shape

    pad_ht = filter_ht // 2
    pad_wdt = filter_wdt // 2
    pad_img = np.pad(I, ((pad_ht, pad_ht), (pad_wdt, pad_wdt)), mode = 'constant')

    im_conv = np.zeros_like(I, dtype = float)

    for i in range(image_ht):
        for j in range(image_wdt):
            roi = pad_img[i:i+filter_ht, j:j+filter_wdt]
            im_conv[i, j] = np.sum(roi * f)
    return im_conv
```

```python
def gaussian_filter(sigma, filter_size):
    # Ensure filter size is odd
    filter_size = filter_size + 1 if filter_size % 2 == 0 else filter_size

    # Calculate range of values
    x = np.arange(-filter_size // 2, filter_size // 2 + 1)

    # Calculate Gaussian kernel
    kernel = np.exp(-0.5 * (x / sigma) ** 2) / (sigma * math.sqrt(2 * math.pi))
    kernel /= np.sum(kernel)

    return kernel
```

```python
def detect_zero_crossings(image,delta):
        res_img = np.zeros_like(image)
        ht, width = image.shape

        for i in range(1, ht - 1):
            for j in range(1, width - 1):
                mid_pixel = image[i, j]
                if mid_pixel > 0:  # If the center pixel is positive
                    neighbor = [image[i-1, j-1], image[i-1, j], image[i-1, j+1],
image[i, j-1], image[i, j+1], image[i+1, j-1], image[i+1, j], image[i+1, j+1]]
                    for x in neighbor:
                        if x >= 0:
                            continue
                        if image[i, j] - x > delta:
                            res_img[i, j] = 255
                            break

        return res_img
```

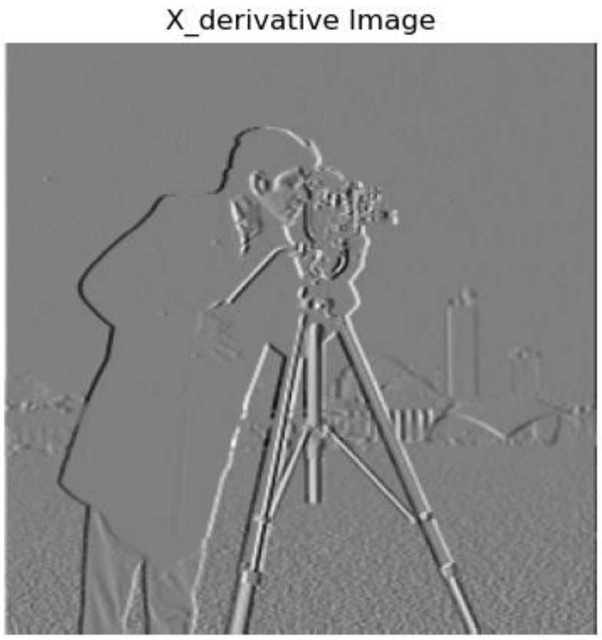
```python
camera = read_image(r"C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer
Vision\Proj\Proj - 2\cameraman.png")
hor_filter = np.array([-1, 0, 1])
ver_filter = np.array([1, 0, -1])

res_hor = convolution(hor_filter.reshape(1, -1), camera)
res_ver = convolution(ver_filter.reshape(-1, 1), camera)

fig = plt.figure(figsize = (10,7))
fig.add_subplot(1,2,1)
plt.imshow(res_hor, cmap='binary')
plt.title("Horizontal edges")
plt.axis("off")

fig.add_subplot(1,2,2)
plt.imshow(res_ver, cmap='binary')
plt.title("Vertical edges")
plt.axis("off")
```

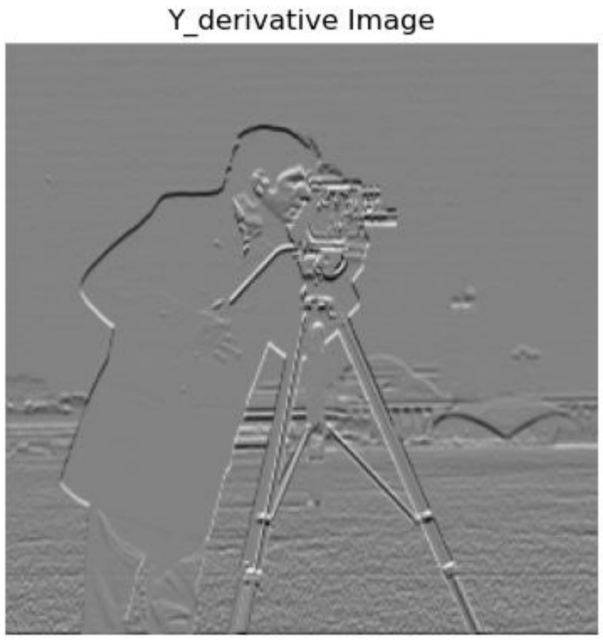




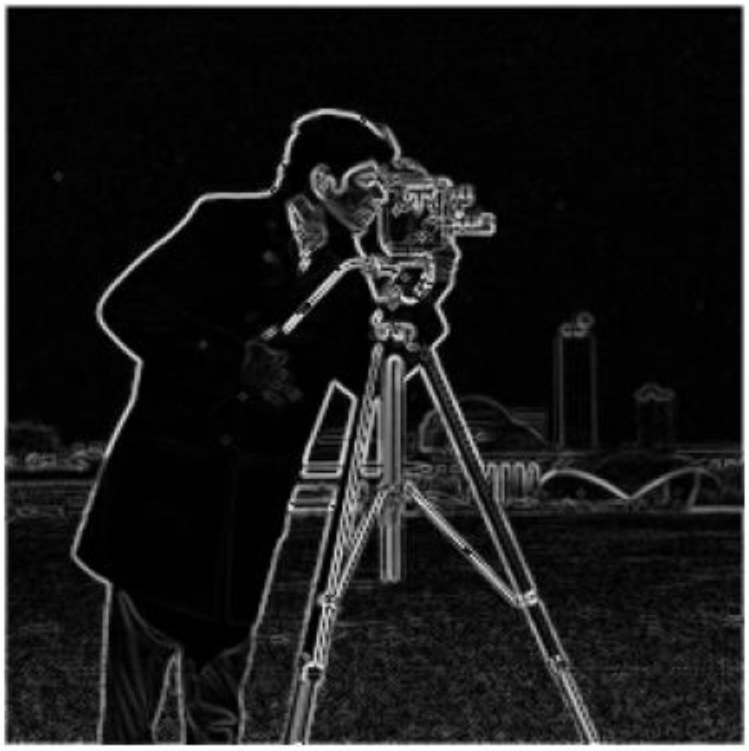
```
dx_filter = np.array([-1, 0, 1]).reshape(1,-1)
dy_filter = dx_filter.T
# print(dy_filter)
```

```
dx_der_img = convolution(dx_filter, camera)
dy_der_img = convolution(dy_filter, camera)

fig = plt.figure(figsize=(10,7))
fig.add_subplot(1,2,1)
plt.imshow(dx_der_img, cmap='gray')
plt.axis("off")
plt.title("X_derivative Image")
cv2.imwrite('Camera_X_Derivative_image.jpg', dx_der_img)

fig.add_subplot(1,2,2)
plt.imshow(dy_der_img, cmap='gray')
plt.axis("off")
plt.title("Y_derivative Image")
cv2.imwrite('Camera_Y_Derivative_image.jpg', dy_der_img)
```



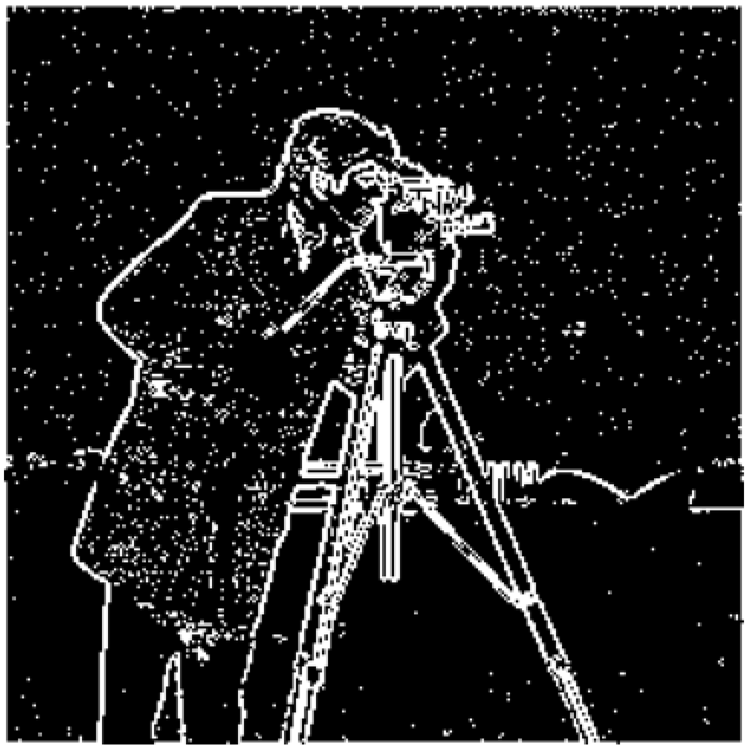


```
grad_magnitude = (np.sqrt(dx_der_img**2 + dy_der_img**2)).astype(np.uint8)
plt.imshow(grad_magnitude, cmap = 'gray')
plt.axis("off")
cv2.imwrite('Gradient_magnitude_Camera.jpg', grad_magnitude)
```

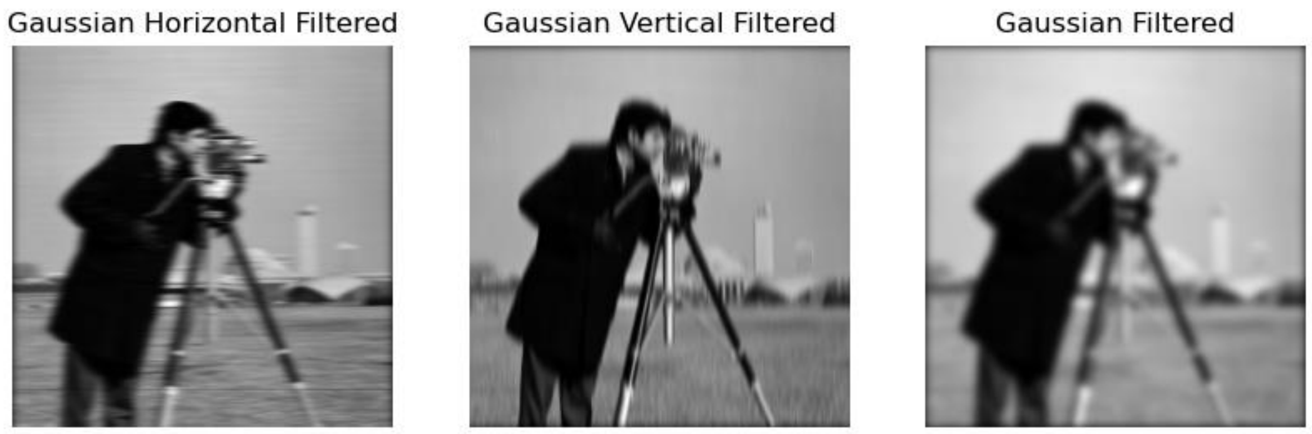


```
threshold, otsu_img = cv2.threshold(grad_magnitude, 0, 255,
cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```

```
bin_edge_img = (grad_magnitude > otsu_img).astype(float)
plt.imshow(bin_edge_img, cmap = 'binary')
plt.axis("off")
```

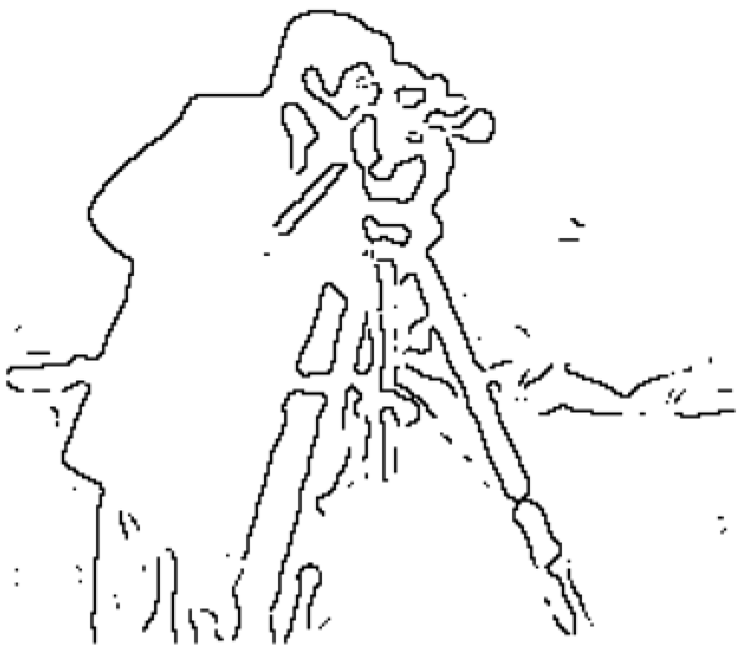


```
laplacian_fil = np.array([[0, -1, 0],
                          [-1, 4, -1],
                          [0, -1, 0]])

sigma = 3.0
filter_size = int(6 * sigma + 1)

Gx = gaussian_filter(sigma, filter_size)
Gy = Gx.reshape(-1, 1)

filt_hor_gau = convolution(Gx[np.newaxis, :], camera)
filt_ver_gau = convolution(Gy, camera)
filtered_img = convolution(Gx[np.newaxis, :], filt_ver_gau)

fig = plt.figure(figsize= (10,7))
fig.add_subplot(1,3,1)
plt.imshow(filt_hor_gau, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Horizontal Filtered")

fig.add_subplot(1,3,2)
plt.imshow(filt_ver_gau, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Vertical Filtered")

fig.add_subplot(1,3,3)
plt.imshow(filtered_img, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Filtered")
```

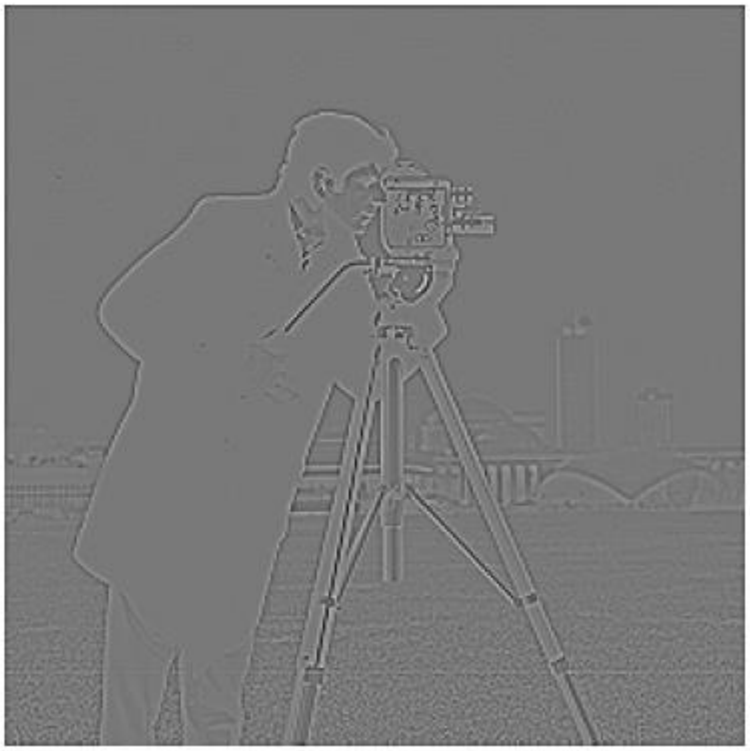


```
lap_img = convolution(laplacian_fil, filtered_img)
plt.imshow(lap_img, cmap='gray')
plt.axis("off")
```

```
res_img = detect_zero_crossings(lap_img, 1)
plt.imshow(res_img, cmap= 'binary')
plt.axis("off")
```

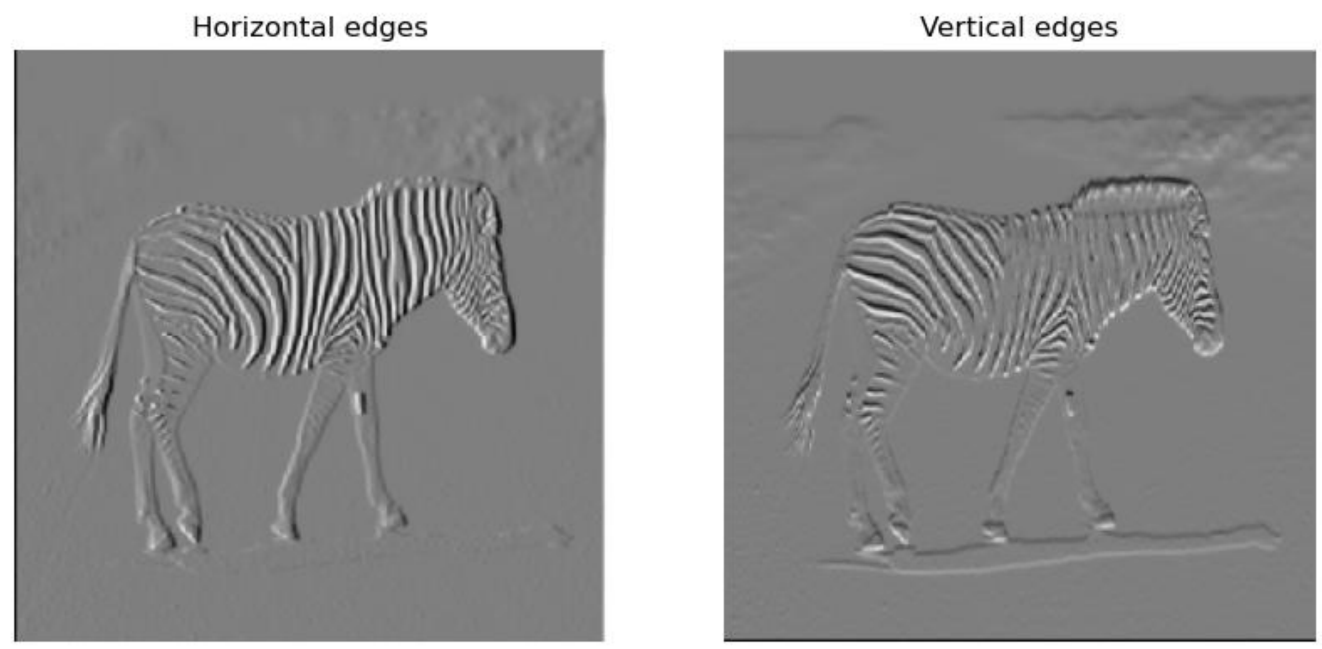


```
res_img_2D_fil = convolution(laplacian_fil, camera)
plt.imshow(res_img_2D_fil, cmap = 'binary')
plt.axis("off")
```

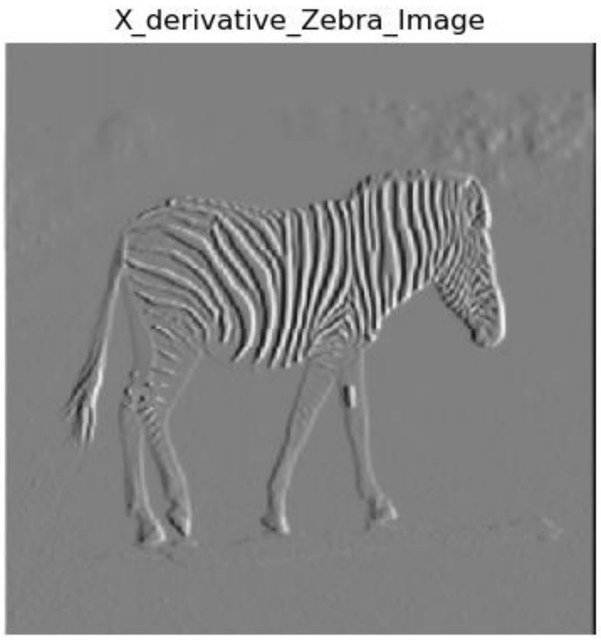


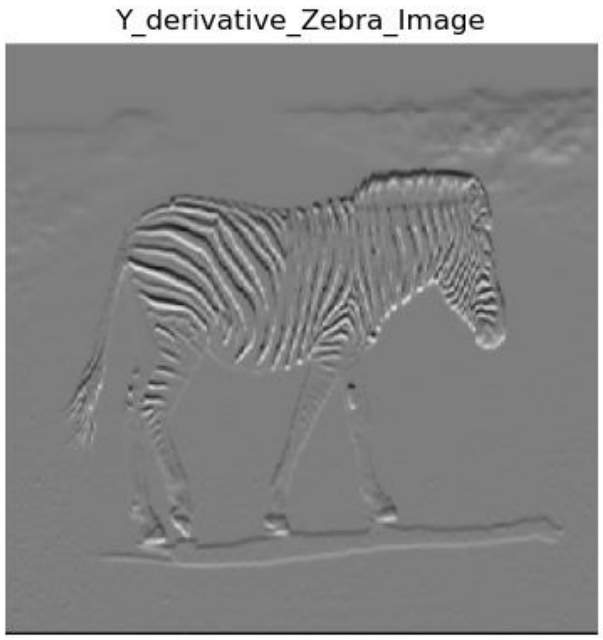
```
zebra = read_image(r"C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer Vision\Proj\Proj - 2\zebra.png")
```

```
hor_filter = np.array([-1, 0, 1])
ver_filter = np.array([1, 0, -1])

res_hor = convolution(hor_filter.reshape(1, -1), zebra)
res_ver = convolution(ver_filter.reshape(-1, 1), zebra)

fig = plt.figure(figsize = (10,7))
fig.add_subplot(1,2,1)
plt.imshow(res_hor, cmap='binary')
plt.title("Horizontal edges")
plt.axis("off")

fig.add_subplot(1,2,2)
plt.imshow(res_ver, cmap='binary')
plt.title("Vertical edges")
plt.axis("off")
```

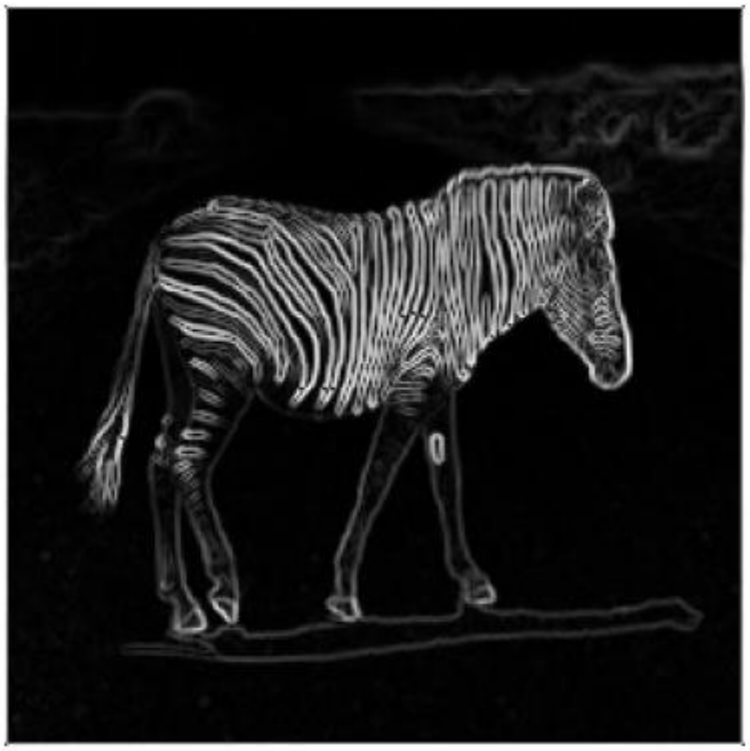


```
dx_der_zebra_img = convolution(dx_filter, zebra)
dy_der_zebra_img = convolution(dy_filter, zebra)

fig = plt.figure(figsize=(10,7))
fig.add_subplot(1,2,1)
plt.imshow(dx_der_zebra_img, cmap='gray')
plt.axis("off")
plt.title("X_derivative_Zebra_Image")
# cv2.imwrite('Camera_X_Derivative_image.jpg', dx_der_img)

fig.add_subplot(1,2,2)
plt.imshow(dy_der_zebra_img, cmap='gray')
plt.axis("off")
plt.title("Y_derivative_Zebra_Image")
# cv2.imwrite('Camera_Y_Derivative_image.jpg', dy_der_img)
```



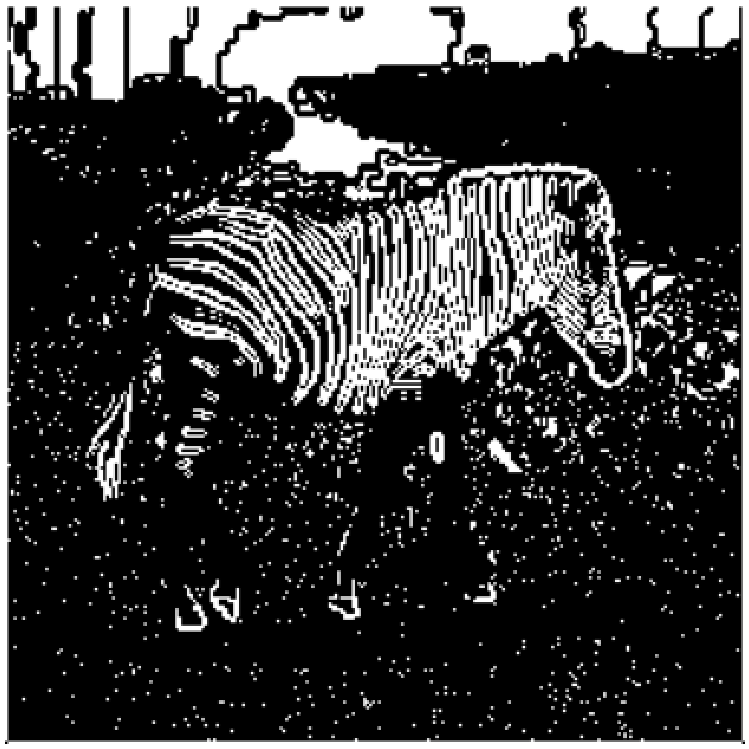


```
grad_magnitude_zebra = (np.sqrt(dx_der_zebra_img**2 +
dy_der_zebra_img**2)).astype(np.uint8)
plt.imshow(grad_magnitude_zebra, cmap = 'gray')
plt.axis("off")
# cv2.imwrite('Gradient_magnitude_Camera.jpg', grad_magnitude)
```

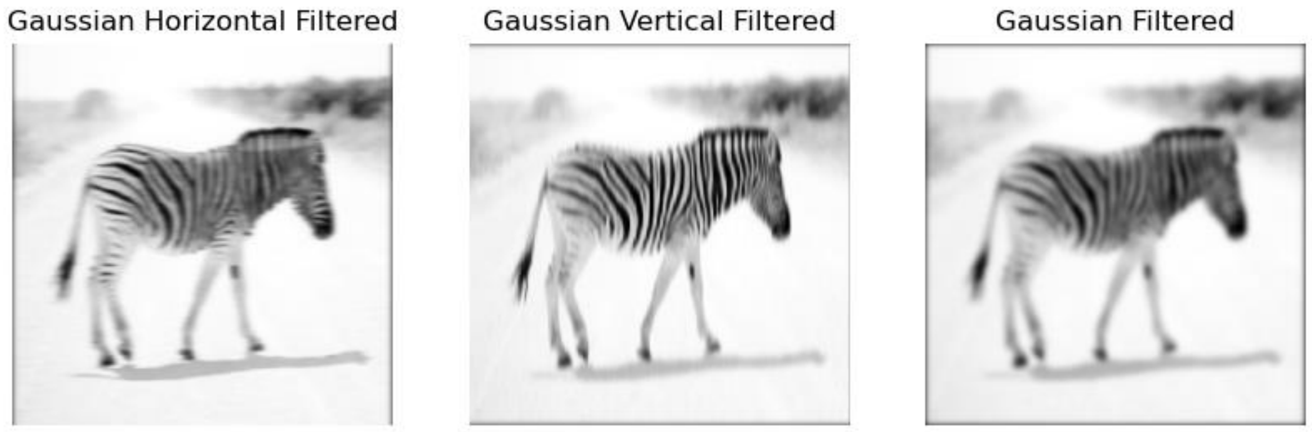


```
threshold, otsu_zebra_img = cv2.threshold(grad_magnitude_zebra, 0, 255,
cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```

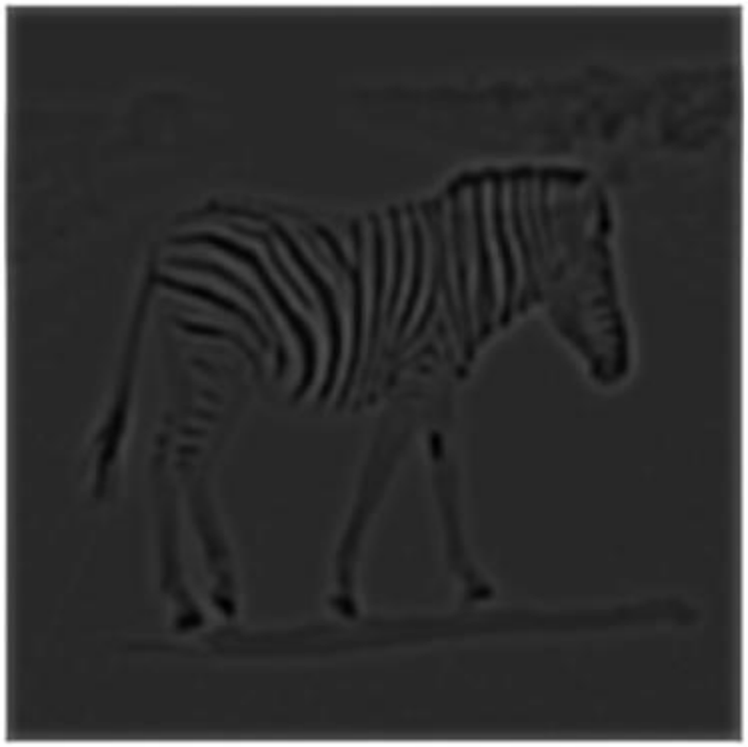
```
bin_edge_zebra_img = (grad_magnitude_zebra > otsu_zebra_img).astype(float)
plt.imshow(bin_edge_zebra_img, cmap = 'binary')
plt.axis("off")
```



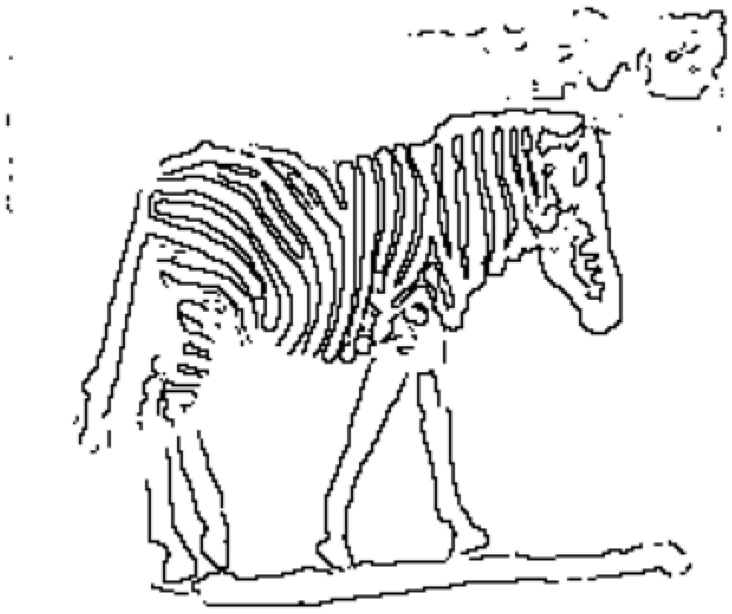
```
laplacian_fil = np.array([[0, -1, 0],
                          [-1, 4, -1],
                          [0, -1, 0]])

sigma_z = 2.0
filter_size = int(6 * sigma + 1)

Gx_z = gaussian_filter(sigma_z, filter_size)
Gy_z = Gx_z.reshape(-1, 1)

filt_hor_gau_zebra = convolution(Gx_z[np.newaxis, :], zebra)
filt_ver_gau_zebra = convolution(Gy_z, zebra)
filtered_zebra_img = convolution(Gy_z, filt_hor_gau_zebra)

fig = plt.figure(figsize= (10,7))
fig.add_subplot(1,3,1)
plt.imshow(filt_hor_gau_zebra, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Horizontal Filtered")

fig.add_subplot(1,3,2)
plt.imshow(filt_ver_gau_zebra, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Vertical Filtered")

fig.add_subplot(1,3,3)
plt.imshow(filtered_zebra_img, cmap = 'gray')
plt.axis("off")
plt.title("Gaussian Filtered")
```



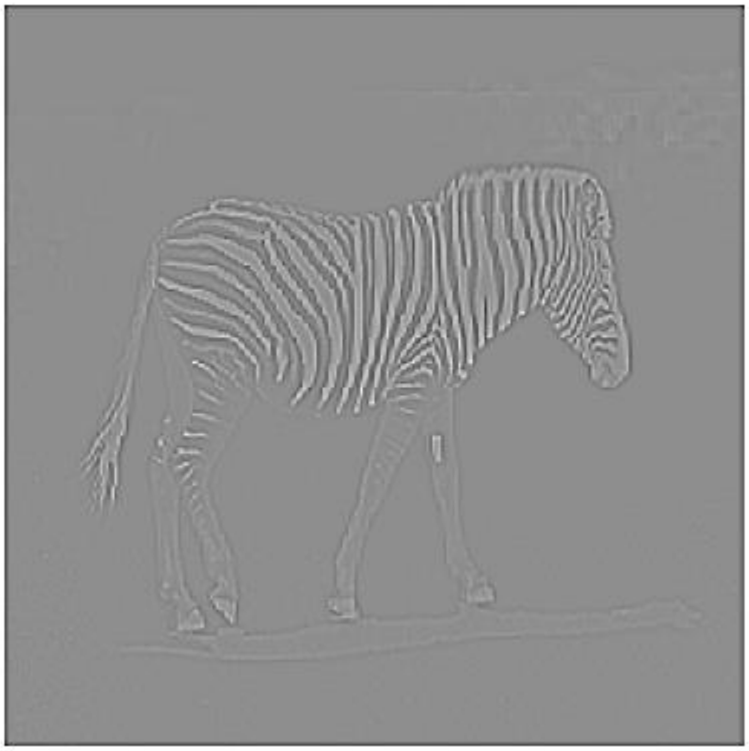
```
lap_zebra_img = convolution(laplacian_fil, filtered_zebra_img)
plt.imshow(lap_zebra_img, cmap='gray')
plt.axis("off")
```



```
res_zebra_img = detect_zero_crossings(lap_zebra_img, 2)
plt.imshow(res_zebra_img, cmap= 'binary')
plt.axis("off")
```



```
res_zebra_img_2D_fil = convolution(laplacian_fil, zebra)
plt.imshow(res_zebra_img_2D_fil, cmap = 'binary')
plt.axis("off")
```



```
def image_adjust(img):
    img_min = np.min(img)
    img_max = np.max(img)
    adjusted_img = 255 * (img - img_min) / (img_max - img_min)
    return adjusted_img.astype(np.uint8)
```

```
def match_template(original_img, template_img):
    # Get dimensions of original and template images
    original_height, original_width = original_img.shape
    template_height, template_width = template_img.shape

    # Compute mean of template image
    template_mean = np.mean(template_img)

    # Compute squared sum of template image
    template_squared_sum = np.sum((template_img - template_mean) ** 2)

    # Initialize correlation image
    correlation_img = np.zeros((original_height - template_height + 1, original_width
- template_width + 1))
```

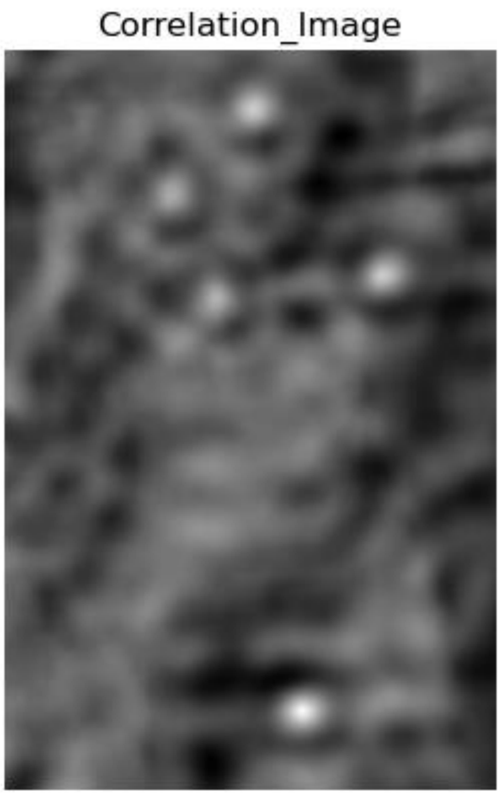
```
    # Iterate through each position in the original image where the template can fit
    for y in range(original_height - template_height + 1):
        for x in range(original_width - template_width + 1):
            # Extract the region of interest (ROI) from the original image
            roi = original_img[y:y + template_height, x:x + template_width]

            # Compute mean of ROI
            roi_mean = np.mean(roi)

            # Compute correlation between ROI and template
            correlation = np.sum((roi - roi_mean) * (template_img - template_mean))

            # Compute squared sum of ROI
            roi_squared_sum = np.sum((roi - roi_mean) ** 2)

            # Normalize the correlation value
            correlation /= np.sqrt(roi_squared_sum * template_squared_sum)

            # Store the correlation value in the correlation image
            correlation_img[y, x] = correlation

    return correlation_img
```

```
def plot_image(img, map, title):
    plt.imshow(img, cmap = map)
    plt.axis("off")
    plt.title(title)
```

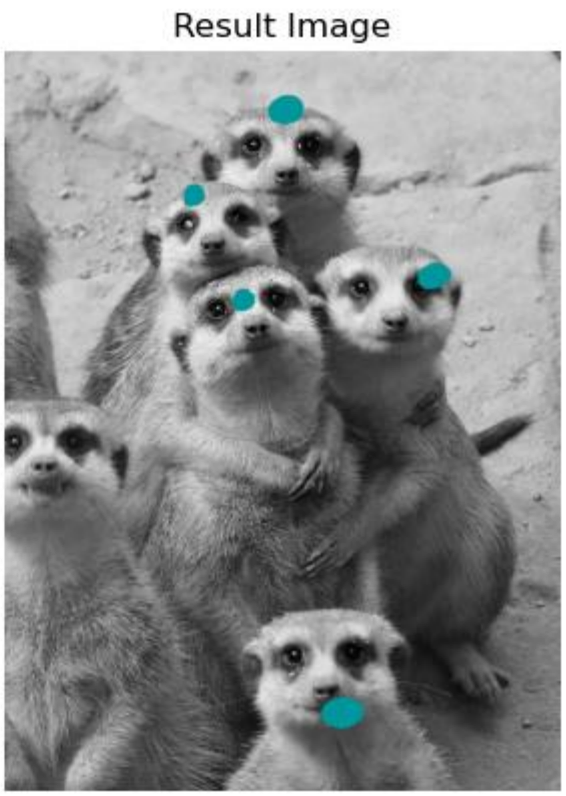
```
original_img = cv2.imread(r'C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer Vision\Proj\Proj - 2\animal-family-25.jpg', cv2.IMREAD_GRAYSCALE)
template_img = cv2.imread(r'C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer Vision\Proj\Proj - 2\animal-family-25-template.jpg', cv2.IMREAD_GRAYSCALE)

    # Convert images to float32
original_img = original_img.astype(np.float32)
template_img = template_img.astype(np.float32)

    # Calculate mean value of the template image
mean_val = np.mean(template_img)

    # Normalize the template image by subtracting the mean
zero_mean_template = template_img - mean_val
```

```
    # Explicitly set the mean of the resulting template to 0.0
zero_mean_template -= np.mean(zero_mean_template)

plot_image(zero_mean_template, 'gray', 'Zero_mean_template')
```

```
# correlation_img = cv2.matchTemplate(original_img, zero_mean_template,
cv2.TM_CCOEFF_NORMED)
correlation_img = match_template(original_img, zero_mean_template)

    # Apply ImageAdjust
correlation_img = image_adjust(correlation_img)
plot_image(correlation_img, 'gray', 'Correlation_Image')
```

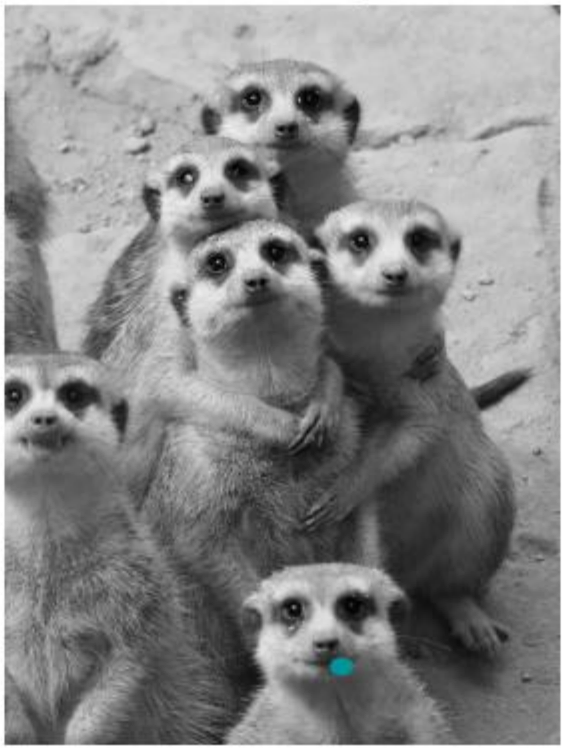


```
threshold = 0.6 * 255
_, peaks = cv2.threshold(correlation_img, threshold, 255, cv2.THRESH_BINARY)

    # Ensure peaks has the same shape as the original image
peaks = cv2.resize(peaks, (original_img.shape[1], original_img.shape[0]))

    # Overlay the peaks onto the original image
overlay = cv2.cvtColor(original_img.astype(np.uint8), cv2.COLOR_GRAY2BGR)
overlay[peaks > 0] = [1, 150, 155]
plot_image(overlay, 'gray', 'Result Image')
```



```
threshold = 0.8 * 255
_, peaks = cv2.threshold(correlation_img, threshold, 255, cv2.THRESH_BINARY)

    # Ensure peaks has the same shape as the original image
peaks = cv2.resize(peaks, (original_img.shape[1], original_img.shape[0]))

    # Overlay the peaks onto the original image
overlay = cv2.cvtColor(original_img.astype(np.uint8), cv2.COLOR_GRAY2BGR)
overlay[peaks > 0] = [1, 150, 155]
plot_image(overlay, 'gray', 'Result Image')
```

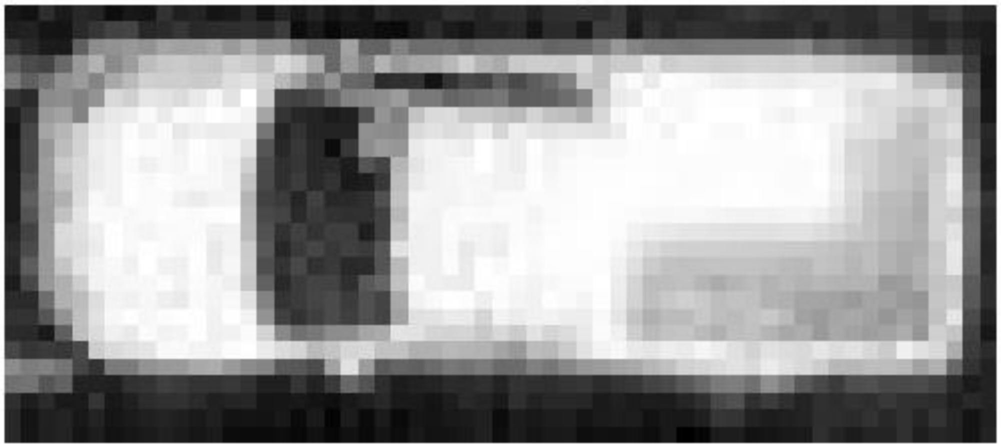
Result Image



```
threshold = 1 * 255
_, peaks = cv2.threshold(correlation_img, threshold, 255, cv2.THRESH_BINARY)

    # Ensure peaks has the same shape as the original image
peaks = cv2.resize(peaks, (original_img.shape[1], original_img.shape[0]))

    # Overlay the peaks onto the original image
overlay = cv2.cvtColor(original_img.astype(np.uint8), cv2.COLOR_GRAY2BGR)
overlay[peaks > 0] = [1, 150, 155]
plot_image(overlay, 'gray', 'Result Image')
```

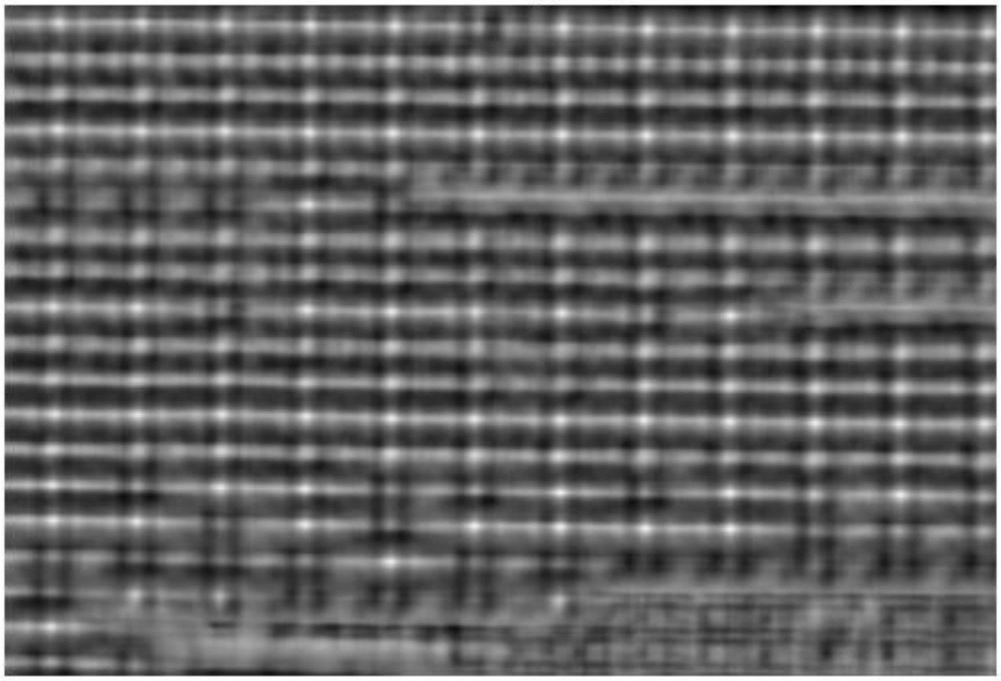
```
original_img = cv2.imread(r'C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer Vision\Proj\Proj - 2\car_full_image.jpg', cv2.IMREAD_GRAYSCALE)
template_img = cv2.imread(r'C:\Users\rhuth\Desktop\SEMS\Sem - 2\CS - GY6643 - Computer Vision\Proj\Proj - 2\car_full_image_template.jpg', cv2.IMREAD_GRAYSCALE)

    # Convert images to float32
original_img = original_img.astype(np.float32)
template_img = template_img.astype(np.float32)
original_img_2 = original_img.astype(np.float32)


    # Calculate mean value of the template image
mean_val = np.mean(template_img)

    # Normalize the template image by subtracting the mean
zero_mean_template = template_img - mean_val

    # Explicitly set the mean of the resulting template to 0.0
zero_mean_template -= np.mean(zero_mean_template)

    #show_image('res 1',zero_mean_template)
plot_image(zero_mean_template, 'gray', 'Zero Mean Template')
```

Zero Mean Template



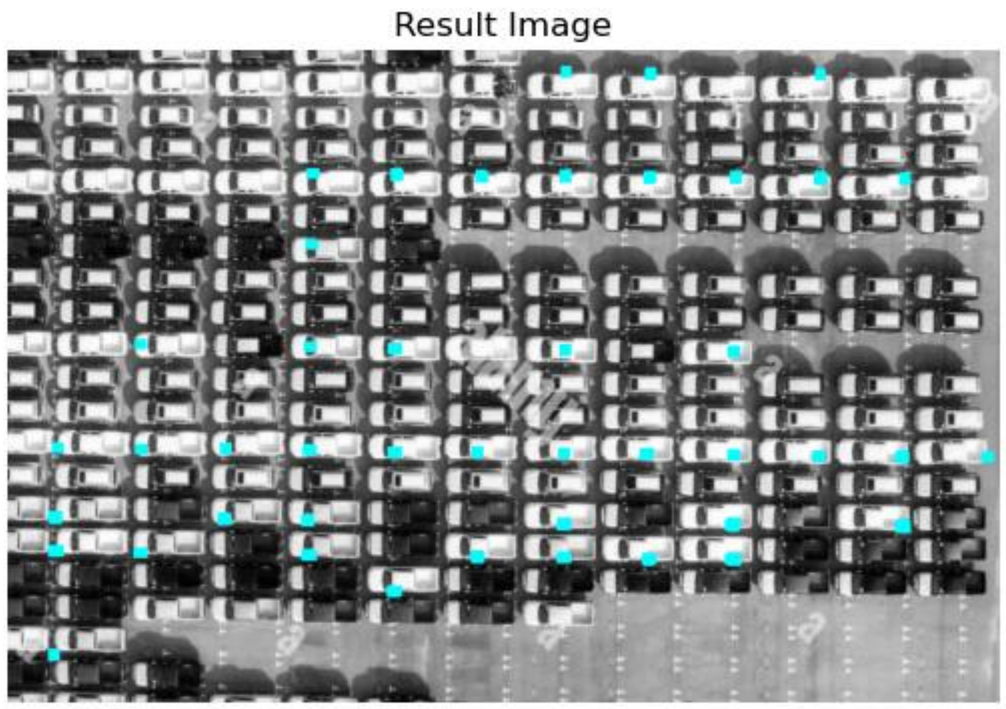
```
correlation_img = match_template(original_img, zero_mean_template)

    # Apply ImageAdjust
correlation_img = image_adjust(correlation_img)
plot_image(correlation_img, 'gray', 'Correlation_Image')
```

Correlation_Image



```
threshold = 0.9 * 255
_, peaks = cv2.threshold(correlation_img, threshold, 255, cv2.THRESH_BINARY)

peaks = cv2.dilate(peaks, None, iterations=3)
```

```
    # Find contours of peaks
contours, _ = cv2.findContours(peaks, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Ensure peaks has the same shape as the original image
peaks = cv2.resize(peaks, (original_img.shape[1], original_img.shape[0]))
    # Overlay the peaks onto the original image
overlay = cv2.cvtColor(original_img.astype(np.uint8), cv2.COLOR_GRAY2BGR)
overlay[peaks > 0] = [0, 255, 255]
plot_image(overlay, 'gray', "Result Image")

print('Number of Cars is: '+str(len(contours)))
```



```
Number of Cars is: 44
```